软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(9):2036−2048 [doi: 10.3724/SP.J.1001.2011.03874]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于信息流策略的污点传播分析及动态验证*

黄 强[1,2+], 曾庆凯[1,2,3]

[1](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

[2](南京大学 计算机科学与技术系,江苏 南京 210093)

[3](上海市信息安全综合管理技术研究重点实验室,上海 200240)

Taint Propagation Analysis and Dynamic Verification with Information Flow Policy

HUANG Qiang[1,2+], ZENG Qing-Kai[1,2,3]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[3](Shanghai Key Laboratory of Integrate Administration Technologies for Information Security, Shanghai 200240, China)

+ Corresponding author: E-mail: qianghuang87@gmail.com

Huang Q, Zeng QK. Taint propagation analysis and dynamic verification with information flow policy. *Journal of Software*, 2011,22(9):2036−2048. http://www.jos.org.cn/1000-9825/3874.htm

Abstract: In this paper, based on a flow and context-sensitive SSA (static single assignment) information-flow analysis, a fine-grained and scalable approach is proposed for taint propagation analysis, which can not only track tainted data and its propagation path with control and data-flow properties, but also detect the vulnerabilities such as buffer overflow and format string bugs successfully. During the analysis, pieces of code considered vulnerable are instrumented with dynamic verification routines, so that runtime security is guaranteed in the absence of user intervention. The analysis system is implemented as an extension of GCC compiler, and the experiments have proven that this approach is efficient, holding both optimized accuracy and time-space cost.

Key words: vulnerability; information flow; taint propagation; dynamic verification; static single assignment

摘 要: 基于流和上下文敏感的 SSA(static single assignment)信息流分析技术,提出了一种细粒度、可扩展的污点传播检测方法.利用控制流和数据流的相关信息,跟踪污染数据及其传播路径,可以检测缓冲区溢出、格式化串漏洞等程序脆弱性.分析过程在潜在问题点自动插装动态验证函数,在无需用户干预的情况下保证了程序的运行时安全.在 GCC 编译器的基础上实现了分析系统,实验结果表明,该方法具有较高的精确度和时空效率.

关键词: 脆弱性;信息流;污点传播;动态验证;静态单一赋值

中图法分类号: TP311 **文献标识码**: A

脆弱性(vulnerability)是指软件中的一个可以被用来恶意取得系统或网络控制权的缺陷或者错误[1].据

* 基金项目: 国家自然科学基金(60773170, 60721002, 90818022, 61021062); 国家高技术研究发展计划(863)(2006AA01Z432); 高等学校博士学科点专项科研基金(200802840002); 江苏省科技支撑计划(BE2010032); 上海市信息安全综合管理技术研究重点实验室开放课题(AGK2008003)

收稿时间: 2010-01-04; 修改时间: 2010-03-39; 定稿时间: 2010-05-05

2008 年 CVE 记录的脆弱性类型分布情况统计,包括 XSS,CSRF,Path Traversal,SQL Injection 和 Format String Vulnerability 等在内的注入型脆弱性(injection vulnerabilities)[2]的比例较大,占到当年总量的 2/3 左右[3].注入型脆弱性与接收外部输入的行为密切相关,使用了未经验证的外部输入数据是脆弱性产生及利用的主要原因[4].外部输入数据亦被称为污染数据(tainted data),通常来源于外部非可信实体的数据源输入,而追踪和预防其不正确使用的过程,也被称为污点传播分析(taint propagation analysis)问题[5].从诱发原理和检测实践上来看,利用污点传播分析来检测和消除程序中的注入型脆弱性是可行的.

目前的污点传播可以分为静态分析(static analysis)和动态分析(dynamic analysis)两类.前者从源代码中抽取语法、语义特征,记录数据流向,判断污染数据的产生、传递和使用;而后者则在程序执行时即时检测数据是否来自外部的非可信输入.静态分析涉及数据流和控制流等多个方面,不需要执行程序.然而,由于动态数据的不可判定性以及源代码语义的描述能力有限,其面临路径爆炸问题[6],分析的精度也一直难以提高.动态方法具有多种脆弱性类型的检测范围,但每次只能检测程序执行时的一条路径,额外的检测操作对程序性能也有着较大影响.例如,Panorama 平均情况下增加的运行开销竟然达 20 倍左右[7].

结合静态和动态分析的特征,本文提出了一种编译级的软件脆弱性检测和预防的解决方案.在我们的方法中,各种语言程序被转换到统一的中间表示形式.静态分析排除不可达执行,搜索导致潜在脆弱性的所有程序路径;以简单形式化规则描述的污点检查策略,易于理解和后续扩充,可识别多种脆弱性类型;最后,在潜在问题点动态插装的数据验证和处理函数,随源代码一起编译链接生成可执行文件,增强了程序的运行时监控能力.我们在 GCC 编译器的基础上实现了基于信息流的污点传播分析系统,针对一些流行的开源程序进行了检测.结果表明,本文的方法可以有效识别和预防污点相关的注入型脆弱性.

本文第 1 节给出一种污点传播分析及动态验证的脆弱性检测方法.第 2 节描述分析系统的设计与实现.第 3 节针对常用开源程序进行了测试,评估分析效果,并演示检测的全过程.第 4 节是相关工作的比较与讨论.最后是全文总结.

1 基于信息流的污点传播分析及验证方法

数据的产生、传递和使用是典型的信息流问题,污点传播也不例外[8].本文的方法以源代码转换得到统一中间表示为基础,主要过程如图 1 所示.首先进行信息流分析,取得程序中变量等实体间的信息流动路径.然后,利用生成的信息流,根据检查策略有选择地识别污染数据,判断其传播范围.在发现污点使用点时,分析程序报告使用位置,并给出数据从污染到使用的整个传播路径.最后,针对潜在问题采取不同的处理办法,默认情况下将由分析程序插入的动态验证函数来保证数据的运行时安全.

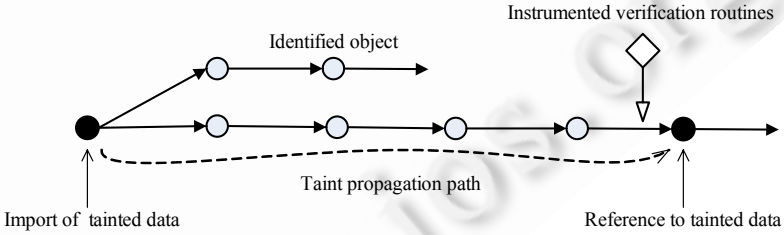


Fig.1 Procedure of taint propagation analysis

图 1 污点传播分析流程

1.1 信息流分析

信息流是信息从一个实体流向另一实体的有效途径,前者称为信息流源头,而后者称为信息流目的地[9].本文将信息流定义为 R,是信息流识别对象集 W 及其上的二元关系$\rightarrow$.

定义 1. 信息流 $R=(W,\rightarrow)$,其中,$\rightarrow=\{\langle src,dest\rangle|src,dest\in W\}$.

识别对象集 W 是广义的概念,包括程序中的变量集 *VARS*、常量集 *CST* 以及函数调用和返回语句等在内的

程序行为实体集合 *OPS*.→则由信息流源对象 *src* 和目的对象 *dest* 组成,多条信息流构成了信息流集合 *IFs*.根据程序控制结构和语法特征,有以下的信息流生成规则:

- 赋值语句

①	$v_0=v_1$;	(v_0,v_1 都不是指针变量)	$IFs=\{\langle v_1,v_0\rangle\}$
②	$v_0=v_1$;	(v_0,v_1 都是指针变量)	$IFs=\{\langle v_1,v_0\rangle,\langle v_0,v_1\rangle\}$
③	$v_0=\&\ v_1$;	(&是取地址操作)	$IFs=\{\langle v_1,v_0\rangle,\langle v_0,v_1\rangle\}$
④	$v_0=a[i_1]...[i_n]$;	(a 为 N 维数组元素)	$IFs=\{\langle a[i_1]...[i_n],v_0\rangle,\ldots,\langle i_1,v_0\rangle,\langle i_n,v_0\rangle\}$
⑤	$v_0=\odot(v_1,\ldots,v_N)$;	($\odot$是 N 元运算符)	$IFs=\{\langle v_1,v_0\rangle,\ldots,\langle v_N,v_0\rangle\}$
⑥	$v_0=\varphi(v_1,v_2,\ldots,v_N)$;	(φ是 SSA 中φ-函数)	$IFs=\{\langle v_1,v_0\rangle,\ldots,\langle v_N,v_0\rangle\}$

- 函数调用语句

①	$fun(v_1,\ldots,v_N)$;	(*fun* 是调用的函数名)	$IFs=\{\langle v_1,fun\|1\rangle,\ldots,\langle v_N,fun\|N\rangle\}$

- 返回语句

①	return v_0;	$IFs=\{\langle v_0,*\rangle\}$

注:*是当前函数的调用者.

- 控制语句

①	if (v_0==k)		{v_1=…}	
	else		{v_2=…}	$IFs=\{\langle v_0,v_1\rangle,\langle v_0,v_2\rangle\}$
②	switch (v_0) {	case i:	{v_1=…}	
	…	default:	{v_N=…}	
	}			$IFs=\{\langle v_0,v_1\rangle,\ldots,\langle v_0,v_N\rangle\}$
③	while (v_0==k)		{v_1=…}	$IFs=\{\langle v_0,v_1\rangle\}$

对于函数调用,需要记录参数的使用顺序,如$\langle v_i,fun|i\rangle$是 v_i 流向函数 *fun* 的信息流,*fun*|*i* 表示 v_i 作为函数的第 *i* 个实参使用.本文采用了无干扰(noninterference)[10]的概念,上述按规则生成的是直接信息流,是原子的,信息由源对象直接流向目的对象.直接流中又包括显式(explicit)和隐式(implicit)信息流[9].前者反映的是实体间的数据依赖,后者反映了实体和程序控制逻辑间的关系.例如,选择语句中作为判断条件的变量,其值决定了条件分支的执行路径,由此生成隐式信息流.直接信息流是通过程序实体间的赋值、运算和控制分支选择等操作而产生的,在识别时将其与产生信息流向行为的程序语句关联在一起,以此标识出信息流的作用范围.信息流还具有传递性的特征,多条直接流的中介和数据转移构成的信息传递路径称为间接信息流[11].间接流以路径的起点和终点作为源和目的对象,其内部记录的路径点为信息传递时经过的实体,而且任意两个相邻的路径点都可以还原为直接信息流.

$$IN(n)=\begin{cases}\varnothing, & n=entry(Blocks(G))\\ \cup\{OUT(i)\mid i\in prevs(n)\}, & n\neq entry(Blocks(G))\end{cases} \tag{1}$$

$$OUT(n)=TRAN(IN(n)-KILL(n),GEN(n))\cup GEN(n)\cup IN(n)\ n\in Blocks(G) \tag{2}$$

信息流分析在程序控制流图(control flow graph)*G* 的基础上进行.计算各基本块的信息流生成集合 *GEN*,销毁集合 *KILL*,传递集合 *TRAN* 以及流入流出基本块的集合 *IN* 和 *OUT* 间的关系,见信息流方程(1)、方程(2)[12].*Blocks*(*G*)是 *G* 中基本块,*entry*()取入口块,*prevs*()取其前驱,*TRAN*(s_1,s_2)是对 s_1 和 s_2 中的信息流做传递操作,得到间接信息流,为程序中信息传递路径的集合.方程迭代计算至不动点之后,反映的是程序全局范围内的可达性关系,即源和目的实体间完整的信息流动路径.

1.2 污点传播模型

污染数据是外部非可信实体的输入所致,在某些数据访问和函数调用时引起潜在的安全问题.因此,在信息流分析已经取得程序中信息传递路径的前提下,污点传播分析的任务就简单了许多,重点是对污染数据产生和安全相关引用的仲裁,其充当的是一个检验器(checker)的角色.

Denning 在程序语义的基础上建立了基于格(lattice)的信息流安全模型[9],本文将此扩展到了污点传播的检

测领域,用一个污点传播安全格$\langle L,\cap,\cup\rangle$来表示分析对象的安全级.

定义 2. 代数格结构 L 由元素集合和两个运算$\cap$(交)、$\cup$(并)组成,满足下面的性质:

(1) 对所有的 $x,y\in L$,存在唯一的 $z\in L,w\in L$,使得 $x\cap y=z,x\cup y=w$(封闭性);

(2) 对所有的 $x,y\in L,x\cap y=y\cap x$,并且 $x\cup y=y\cup x$(交换性);

(3) 对所有的 $x,y,z\in L,(x\cap y)\cap z=x\cap(y\cap z)$,并且$(x\cup y)\cup z=x\cup(y\cup z)$(结合性);

(4) L 中存在两个唯一的元素$\perp$和 T,使得所有 $x,y\in L,x\cap\perp=\perp,x\cup\mathsf{T}=\mathsf{T}$.

定义 3. 在格 L 上定义了偏序关系$x\leqslant y$,当且仅当 $x\cap y=x$,满足下面的性质:

(1) 对所有的 x,有 $x\leqslant x$(自反性);

(2) 对所有的 x,y,如果 $x\leqslant y,y\leqslant x$,则 $x=y$(反对称性);

(3) 对所有的 x,y,z,如果 $x\leqslant y,y\leqslant z$,有 $x\leqslant z$(传递性).

数据在污点传播问题上表现为 untainted(清洁)、tainted(污染)和 vulnerable(脆弱)3 种安全级,满足$\perp\leqslant untainted\leqslant tainted\leqslant vulnerable$,如图 2(a)所示.$\perp$表示安全级未知,为变量的初始安全级.$L$ 上还定义了由程序变量到安全级的映射 $level:VARS\rightarrow L$,取数据在当前程序环境下的安全级.

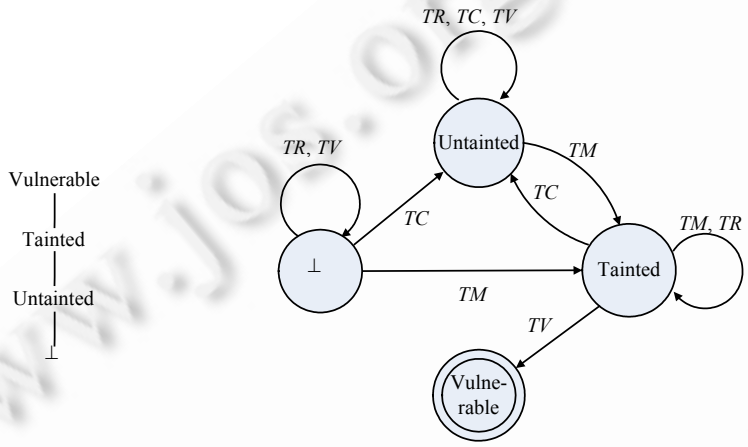


Fig.2 Lattice model and security state transitions in taint propagation analysis

图 2 污点传播分析中的格模型及安全状态转换

本文将污点传播模型定义为四元组:$M=(R,L,T,f)$.

- R 为程序中信息流关系;
- L 为污点传播的安全格;
- T 是安全级的传播模式;
- $f:VARS\times OPS\rightarrow L$ 是一个映射,定义了程序变量及作用于其上的行为实体到变量安全级的转换关系.

W 中有 $VARS\subseteq W$ 且 $OPS\subseteq W$,安全级在信息流中通过赋值、运算以及控制分支的选择等操作,遵循定义 4 的模式 T,在变量间进行传播,而在受到程序行为作用时发生改变.OPS 中存在 4 种程序行为实体,分别对应污点传播生命周期中污染(产生)、清除、传递和安全操作使用这 4 个动作.因此,映射 f 中的转换规则见表 1 的 4 种形式.

定义 4. 安全级在信息流 r 中传播,当且仅当:$\exists v_1,v_2\in obj(r),v_1,v_2$ 在 r 中按传递顺序出现,有 $v_1\in VARS$ 且 $v_2\in VARS$,使得 $level(v_2)=level(v_1)$.其中,$obj(r)\subseteq W$,取信息流 r 中的识别对象:对于直接信息流,仅返回源和目的对象;间接信息流则按程序执行时的信息流动顺序,从源至目的对象,取出其信息传递路径上的所有实体.

Table 1 Rules for security state transitions and classification of operation entities

表 1 污点传播安全级转换规则及行为实体分类

Rule	Role of entity	State transition	Comment
Taint rule *R*1	*TM* (taint-maker)	$f(m,n)=tainted$, $m\in VARS\wedge n\in TM$	Functions or operations that import tainted data are the sources of taint
Clean rule *R*2	*TC* (taint-cleaner)	$f(m,n)=untainted$, $m\in VARS\wedge n\in TC$	Operations that reset the data field make tainted data de-taint
Propagation rule *R*3	*TR* (taint-relay)	$f(m,n)=level(m)$, $m\in VARS\wedge n\in TR$	Taint irrelevant operations just pass on the security states, instead of changing them
Sink rule *R*4	*TV* (taint-victim)	$f(m,n)=vulnerable$, $m\in VARS\wedge level(m)=tainted\wedge n\in TV$	Vulnerabilities occurred when security operations referred to the tainted data

4 种程序行为实体对数据的影响构成了各安全级间的转换,得到如图 2(b)所示的转换图.从初始状态⊥开始,清洁的数据经过 *TM* 变为污染的,*TV* 使用了污染数据将导致脆弱性的产生.与传统的污点传播格模型[13]相比,3 种安全级细化了数据的污染状态转换粒度,新增的 *vulnerable* 更为符合脆弱性产生的实际情况,只有当安全操作引用了污染数据之后才有可能导致潜在的脆弱性.因此在我们的模型中,污点传播分析的检查点是所有的安全操作,即上述 *TV* 类型的程序行为实体.而安全策略的定义为:程序在到达检查点时,没有产生违反污点传播安全格 *L* 中偏序关系定义的信息流,不会导致触发规则(sink rule)*R*4 的转换发生.

污点传播分析不仅要关注污染数据的产生和使用位置,更重要的是找出污点传播的路径和影响范围.信息流反映了程序实体间的信息流动,其内部记载了信息传递时经过的实体.因此,基于信息流的污点传播判定过程可以定义为如下的形式化描述.

定义 5. 信息流 *r* 传播了污染,导致潜在的脆弱性,当且仅当:$\exists p_1,\ldots,p_i,\ldots,p_n\in obj(r)$, $p_1,\ldots,p_i,\ldots,p_n$ 在 *r* 中按传递顺序出现,有 $f(p_1,p_2)\subseteq R1\wedge f(p_i,p_{i+1})\not\subset R2\wedge f(p_{n-1},p_n)\subseteq R4$.

信息流 *r* 中表示路径安全级的状态变量的初始时为⊥,根据传播模式 *T* 和转换规则 *f*,如果有 *TM* 导致的污染数据流向了 *TV*,而且 *TC* 没有出现在其传递路径上,就可以判定其传播了污染,导致潜在的脆弱性.

1.3 污点检查策略

制定污点传播的检查策略,就是定义程序的可信边界(trusted-boundary).可信边界是系统不受外界影响的数据处理边界[14],边界的一端,数据是不可信的;而另一端,数据对特定操作是安全的.可信边界的入口被称为程序的攻击面(attack surface)[15],后者被定义为一个程序接受输入数据的接口集,一般由程序的入口点和外部函数调用所组成,这构成了划分污点传播模型中程序行为实体的基本依据.

检查策略需要提供污点传播模型中元素的识别判定规则,包括指定上述 4 种程序行为实体,以及对 *TV* 的脆弱性加以分类,提示有效的修正信息.本文认为,程序行为实体并非单纯的函数或访问操作,而是函数和特定参数,以及操作和特定数据的组合.例如,外部输入数据保存在变量 *buf* 中,有如下的调用:"*printf*(*buf*);".如果 *buf* 包含"%s"等格式化符号,就有可能导致格式化串漏洞.相反,在程序已提供正确格式化符号的前提下,其类型和个数与栈中参数相匹配,即使引用了污染数据,也不会出现问题.

本文将污点检查策略表示为一个由实体类型 *type*、脆弱性描述 *vul*、程序操作 *op* 以及操作数位置 *loc* 组成的四元组:

$$\langle type,vul,op,loc\rangle \mid type\in ROLES, vul\in VUL\_TYPES, op\in ACTS, loc\in\{N\cup any\},$$

其中:*ROLES* 是模型中程序行为实体的类型集合,包括 *TM*,*TC*,*TR* 和 *TV* 这 4 个元素;*ACTS* 是程序行为集合,包括函数调用和数组访问操作等;*VUL_TYPES* 和 *N* 则分别为脆弱性类型以及自然数集合;符号 *any* 表示程序操作中的任意操作数位置.

VUL_TYPES 定义了分析程序能够识别的脆弱性类型.本文考虑了表 2 中的 6 种分类,其中,有的由隐式流产生,如 *setjmp*/*longjmp* 函数,可能受到外部条件变量的影响,在分支中选择执行.该类函数主要用于 C 语言的错误恢复,在其他语言中演变成更为普通的异常处理机制.虽然不很常用,但还是有检测的必要.

Table 2 Types for *TV* vulnerability entity

表 2 *TV* 类型实体脆弱性分类

Vulnerability	*VUL_TYPES* tag	Example functions or operations
Buffer overflow (array access violation included)	*BUFFER_OVERFLOW*	Buffer manipulation functions (e.g. strcpy, strcat) and array access operations
Memory fault caused by integer overflow	*MEMORY_OVERFLOW*	Functions for memory allocation (e.g. *mallco*, *calloc*) that need unsigned and non-negative integer arguments
Format-String vulnerability	*FORMAT_STRING*	The printf family of functions (e.g. *printf*, *sprintf*, *vprintf*)
Path traversal	*PATH_TRAVERSAL*	Functions for file manipulation (e.g. *open*, *chdir*, *chmod*) that need path description arguments
Jump control-flow hijack	*JUMP_HIJACK*	Some control-flow transfer functions (e.g. *setjmp*, *longjmp*, *sigsetjmp*)
Malicious execution	*MALICIOUS_EXC*	The exec family of functions (e.g. *execl*, *execlp*, *execle*)

本文以 C 语言程序库函数和 Linux 系统调用为目标实例,分类并列举了其中部分典型的函数和操作,见表 3.⟨*TM*,*NULL*,*main*,2⟩为系统 *main* 函数的第 2 个参数引入了污染,⟨*TV*,*FORMAT_STRING*,*printf*,2⟩表示以外部数据作为第 2 个参数调用 *printf*,可能导致格式化串漏洞,而⟨*TV*,*BUFFER_OVERFLOW*,*ArrayAcces*,*any*⟩表示的是以外部数据作为任意维下标访问数组元素可能引起缓冲区溢出.

Table 3 Examples of operation entities in taint propagation analysis (partial)

表 3 污点传播模型中的行为实体分类举例(部分)

Role of entity	Example functions or operations
Taint-Maker	⟨*TM*,*NULL*,*sacnf*,2⟩,⟨*TM*, *NULL*,*receive*,2⟩,⟨*TM*,*NULL*,*main*,2⟩,…
Taint-Cleaner	Functions or operations like regular expression matching, make the tainted data clean. We do not take them into account in C programs for the present
Taint-Relay	Taint irrelevant functions or operations would be found easily once the other 3 kinds of entities were determined
Taint-Victim	⟨*TV*,*FORMAT_STRING*,*printf*,2⟩,⟨*TV*,*BUFFER_OVERFLOW*,*strcpy*,2⟩, ⟨*TV*,*BUFFER_OVERFLOW*,*ArrayAcces*,*any*⟩,…

1.4 污染数据的动态验证

现代程序使用了大量来自外部的输入数据,对每个安全相关的污染引用都做出脆弱性警告来要求用户修正是不现实的,其工作量可能相当巨大.脆弱性的产生与污染数据的内容密切相关,不能仅凭污染数据的使用就判断脆弱性的存在与否.

解决问题的办法是将验证过程推迟到运行时进行,我们采用了在分析代码中插装(instrumentation)的思路来执行污染数据的验证操作.信息流分析追踪污染数据的产生、传递和使用,污点传播则判断污染数据是否导致潜在的脆弱性.在二者的静态分析排除了不会导致脆弱性的传递路径之后,分析程序根据脆弱性类型,在找到的潜在位置之前插入了对污染数据的验证函数,运行时决定数据的合法性.

插装的验证函数以定制的方式来实现,允许灵活的动态验证策略,满足不同环境下的性能要求.针对第 1.3 节中的脆弱性类型,表 4 列举了本文实现的 5 种动态验证功能.其中的 *match*()是条件匹配的抽象操作,条件满足时,将执行箭头右边的预定动作.*error_log*()向用户报告一个错误,打印调用堆栈,*abort*()则终止程序的正常执行.控制流劫持和恶意文件执行需要更加复杂的安全机制来处理,在此暂不作考虑.插装框架在设计上是开放的,可以根据程序所处的安全环境,选择合适的验证函数和处理例程.

Table 4 Dynamic verification policies for 5 kinds of vulnerabilities

表 4 5 种脆弱性类型的动态验证策略

Vulnerability	Verification policy	Comment
Buffer overflow	*match*(*LEN*(*dest*)<*LEN*(*src*)||*N*>*LEN*(*src*))→*error_log*(); *abort*();	Length of the destination buffer must be greater or equal than that of the source buffer (e.g. *strncpy*(*dest*, *src*, *N*))
Array access violation	*match*(∃*i*∈*dim*(*ArrayAccess*)|*index*$_i$≥Max_*Dim*$_i$)→*error_log*(); *abort*();	Index used for array access must be less or equal than the maximum of the current dimension
Memory fault caused by integer overflow	*match*((*int*)*unsigned_var*<0)→*error_log*(); *abort*();	The unsigned integer when interpreted as the type of signed should be greater or equal than 0
Format-String vulnerability	*match*("%[\ \-\#\+]*(\d**?)\.?(\d**?)(a|A|c|d|e|E|f|g|G|i|ld|n|o|p|s|u|x|X)")→*error_log*();	Format specifiers (e.g. "%s") should not be allowed in tainted data, by checking with regular expression matching
Path traversal	*match*("\.{1,2}\\")→*error_log*();	Path descriptors (e.g. "..") should not be allowed in tainted data, by checking with regular expression matching

2 系统实现

大多数编译器在词法、语法分析之后都有程序优化的过程,本文将信息流和污点传播分析整合进入 GCC 编译器的优化框架,是个不错的选择.编译器扫描源代码之后,将其转换为统一的中间表示形式,保持了代码的原始风貌,又消除了各种语言间的异构性.

为提高分析精度,本文使用了静态单一赋值 SSA(static single assignment)[16]的中间表示,保留较多的语义信息,而且相对 Augmented SSA[17]具有较小的转换开销.

GCC 分为前端(front-end)、中端(middle-end)和后端(back-end)这 3 部分.前端接收多种语言源文件输入,生成统一的中间表示.中端基于 SSA 的优化框架加入了新的信息流分析,污点传播分析以及动态验证的过程,后两者各以其之前的分析结果为基础,整个分析程序的结构如图 3 所示.

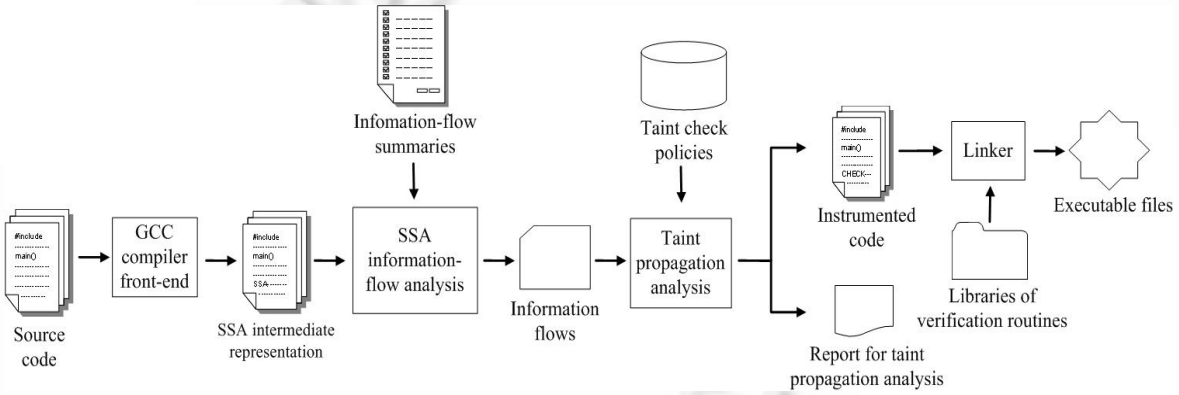


Fig.3 Architecture of taint propagation analysis system

图 3 污点传播分析系统结构图

2.1 SSA信息流分析

SSA 的中间表示将同一变量的多次赋值分离为同源但不同版本的多个实例,其每个变量仅有唯一定值的特性,使得基于此的信息流分析无需计算方程(2)中的销毁集合 KILL,保留语义的基础上简化了分析过程.信息流除了有源和目的对象之外,对于间接流,还要记录传递计算时起中介作用的对象,每条直接流都要绑定到其信息流向行为发生时的语句位置,由间接流传递经过的相邻实体还原后可以作为路径追溯的定位点.我们利用了 GCC 内部的别名计算结果[18,19],可以处理普通的变量指针以及结构化变量的别名问题.信息流实体间的可达性关系在宏观上由基本块间的可达性来反映,位于同一块内的,则由其所处的语句顺序来决定,因此可达性的计算过程是流敏感(flow-sensitive)的.在处理涉及函数调用的信息流时,针对不同的调用点(call-site)识别不同的信息

流对象,分析结果在调用点是上下文敏感(context-sensitive)的.

几乎所有基于源代码的分析技术都会遇到一个不可回避的问题——待分析函数的源代码缺失,在分析库函数以及当前还未编译的函数时尤为如此.最简单的办法就是忽略它们,然而所带来的精度损失也是不能容忍的.这里采用了摘要(summary)的思路,载入预先配置的函数信息流说明,分析时自动匹配,生成缺失的信息流.所有通过参数传递返回值以及参数之间有信息流动的源代码不可见函数,都有为其生成摘要的必要.对于暂时还未编译的函数,则考虑将过程间分析推迟至该函数编译过后再来进行,毕竟基于源代码的即时分析比摘要说明具有更佳的识别范围和处理粒度.

本文以 GCC“unit-at-a-time”[20]的编译方式为基础,实现了过程间的信息流分析框架.它根据函数间的调用关系,将可能跨多个源代码文件的函数组织在一个编译单元(compilation unit)中进行编译和分析.全局的信息流分析与编译单元的生命周期同步.随着编译单元的创建和销毁,分析所需的资源也要做相应地申请和释放.以编译单元为单位的处理方式极大地减轻了分析过程对系统资源的需求程度,将一次分析整个程序的资源需求总量转化为处理(每个)编译单元时的内存需求.因此,分析整个程序所需的空间开销也就转化为处理编译单元所需空间的最大值,而后者是与单元内的信息流数目成正比的.另外,我们还借助了 GCC 垃圾收集器 GGC[21]的帮助完成内存操作,保证了内存申请与释放的安全和高效.

然而,在面对复杂程序时,分析得到的信息流数目仍然会呈指数式增长.为了节约存储空间和信息流检索的方便,应对分析结果做约减处理.信息流的后期处理以需求驱动(demand-driven)原则为指导,程序变量在行为实体的作用下发生安全级的转换,没有 *TM*,*TC* 和 *TV* 这 3 类实体参与的信息流不改变变量的安全级,后续分析中已无保留的必要.

2.2 污点传播分析及动态验证

污点传播分析可以是一个前向(forward)的过程,由 *TM* 开始,寻找其到 *TV* 的传播路径.同时,也可以用后向(backward)的方法来处理,先找到 *TV* 的使用点,再向后追溯其参数是否来源于 *TM*.前者按照程序执行的顺序以及数据传播的方向,比较适合多类型、污点相关的漏洞检测工作;而后者则在特定模式的漏洞匹配上具有更好的针对性.本文采用的是“前向”的分析方法.

从图 2 可以看出,污染数据还有再次变为清洁的可能.然而,C 标准库中缺乏清除污染的相关函数或操作.故本文并未在污点检查策略中定义定义 5 中清除规则(clean rule)*R*2 相关的 TC 类型实体,即假定 *R*2 在 C 语言程序中不会发生,分析时只需满足 *R*1 和 *R*4,就导致了脆弱性.

污点传播分析从约减后的信息流中取出数据传递时经过的实体,按照第 1.2 节定义的污点传播模型执行转换和匹配过程.其间,针对每一条信息流所表示的传递路径,分析程序都将建立和维护一个状态变量,表示当前路径的安全级.状态变量初始时为“⊥”,当前路径处理完之后便可将其释放.最终,污点传播分析的结果是污染数据的传递路径、脆弱性报告以及插装后的源代码.基于扩展性的考虑,污染判定规则都以结构化索引列表的方式来实现,将事先准备好的检查策略作为配置文件存放在分析程序外部,即时载入为分析提供判断的依据.

动态验证的目的旨在减轻用户的修改负担,保证程序运行时的安全.分析程序的插装过程在语法树的层次上完成,在污染使用点之前自动插入了污染数据验证函数的调用语句,不需要修改源文件,亦不会影响正常的编译流程.随后,链接程序(linker)将实现好的、保存在外部链接库中的验证函数和源代码编译得到的目标文件相链接,生成最终的可执行程序.实际应用时,针对具体的性能要求,可以开启或者关闭插装的动态验证功能.

3 实 验

本文选取表 5 的开源程序为测试对象,包括普通应用程序、FTP 服务器和 HTTP 服务器 3 个分组.它们作为广泛使用的应用软件,具有较好的代表性.

Table 5 List of evaluation programs

表 5 测试对象列表

Program	Version	Lines of code (K)	Number of functions	Comment
gzip	1.3.5	10	113	Software used for file compression
muh	2.2a	8.7	140	Irc-Bouncer for unix
cfengine	3.0.2b1	80	940	Configuration management system
bftpd	2.3	5.7	146	Open-Source FTP server
vsftpd	2.1.2	18	585	FTP server for Unix-like systems
wu-ftpd	2.6.0	19	286	Commonly-Used FTP server
corehttp	0.5.3.1	1.3	30	Single-Process TCP/IP HTTP server
lighttpd	1.4.22	55	925	Light-Weight and flexible web server
apache	1.3.4	110	1 065	The most popular HTTP server in use

3.1 性能测试

测试在 Intel Core(TM)2 Duo 2.53GHz CPU,2GB 内存和 CentOS 5.2 Linux 的环境下进行,经统计得到表 6 的分析结果.其中,内存使用是信息流分析在约减前的空间开销,峰值 MAX 表示分析编译单元时的最大值,总和 SUM 是处理所有编译单元的加总值.PT 指路径遍历,SE 指恶意文件执行,IO 是整数溢出引起的内存错误,BO 为缓冲区溢出(包括数组越界),而 FS 和 JMP 分别为格式化串漏洞和 JUMP 语句控制流劫持.

Table 6 Results of information-flow and taint propagation analysis

表 6 信息流和污点传播分析结果

Program	Information-Flow analysis					Taint propagation analysis									Dynamic verification
	Time (s)	Number of flows		Memory (MB)		Time (s)	PT	SE	IO	BO	FS	JMP	Correct	False positive (%)	Number of instrumented
		UnReduced	Reduced	MAX	SUM										
gzip	9.21	33 111	1 337	12	200	2.15	1	0	0	0	1	0	2	0	2
muh	15.16	66 812	4 584	18	473	3.59	0	0	1	20	11	0	21	34	32
cfengine	102.6	237 324	26 472	17	2 027	50.9	89	0	0	120	4	0	91	57	213
vsftpd	21.5	24 097	2 747	16	297	31.7	26	0	4	10	1	0	28	32	41
wuftpd	38.7	52 227	4 698	30	603	33.4	29	0	5	8	5	0	36	23	47
bftpd	177	196 082	1 708	77	1 636	6.01	4	0	0	4	7	0	11	26	15
corehttp	1	4 065	271	2	9	0.5	0	0	0	1	0	0	1	0	1
lighttpd	274	355 209	10 075	92	5 018	36.7	0	0	6	4	1	0	9	18	11
apache	1 123	1 167 630	9 871	148	6 179	73.1	15	5	1	9	1	1	25	21	26

Cfengine,*bftpd*,*lighttpd* 和 *apache* 信息流分析的时间较长,约减之前的信息流数目也很多,主要原因是其程序内部相对复杂的控制结构和较多的信息流识别对象.例如,*cfengine* 通过很多环境变量、全局变量传递数据和作为判断条件改变程序的控制结构,而后三者则具有 MD5 或 SHA1 的校验功能;*apache* 还包含了额外的诸如正则表达式等在内的复杂支持程序库的源代码.上述程序原有的大量识别对象在管理系统配置信息或者计算校验值时,随着 SSA 的变量重命名、各种运算操作以及流和上下文敏感的分析都导致了新的、更多识别对象的产生,信息流可达性计算的时间和空间复杂度也急剧上升.对于普通的应用程序,分析都能够以较小的时空代价完成,而其中 *apache* 的代码规模最大,消耗的系统资源也相对最多.虽然其内存需求总量已经远超出了系统的内存容量,但其峰值仍能保持在系统资源的正常范围内,以编译单元为单位的处理方式使得面向大型复杂程序的分析成为了可能.最后还进行了信息流的约减,删去了与污点传播无关的信息流,缩减了后续分析的问题规模.

污点传播分析是对信息流的仲裁过程,时间开销要少很多,与约减后的信息流数目成正比.从实验结果看出,该算法可以有效检测源代码中预定义的脆弱性类型.由于静态分析的不精确性,误报、漏报的情况难以避免,尤其是检测 *cfengine* 时的误报率相对较高.其作为一个系统管理程序,需要从文件读入大量的配置参数,污染源较多,潜在的污染路径基数也相对较大.在实现中,大量使用全局变量在过程间传递参数,污染的全局变量以及并发条件的不确定性导致了虚假的传播路径.除此之外,表中的误报和信息流分析的准确程度也有一定关系,但更多是源于第 2.2 节简化的污染判定过程,为提高分析效率而忽略了污染清除效果.结果中的漏报则归因于检测规则的完备程度,目前已整理和制定了超过 100 个,包括 C 标准库和 Linux 系统调用等在内的函数或访问操

作的污点检查规则.然而很多安全相关函数的版本众多,如 *malloc*,*xmalloc* 和 *kmalloc* 等,制定详细而全面的检查策略是一个逐步完善的过程.编译器根据链接库信息,可能在中间表示重写调用的函数名,加上库相关的前缀或标记,也间接增加了检测的难度.误报和漏报是矛盾的关系,强化约束条件,误报得以减少,漏报又不可避免;提高建模精度和覆盖面,有效降低了漏报,但又增大了误报的概率.本文的方法针对 C 语言的程序特征,在两者之间做了折衷,兼顾了效率和实用性.

验证操作是保证程序安全运行的必要手段,Huang 等人认为,运行时保护所带来的额外开销属程序必须的范围之内[13].我们对插装的 5 种脆弱性验证函数进行了性能评估,部分结果如图 4 所示.格式化串验证所需的时钟周期随验证数据的大小呈线性递增趋势,数据规模超过 1MB 后趋于平缓,约为 8000.路径和缓冲区验证则保持在相对较小的范围之内,上限分别为 2000 和 1000.整数和下标范围验证属常数时间开销,均小于 100 个时钟周期.从结果来看,验证函数单次执行的开销可视作常量,动态验证的总代价最终和验证函数的执行次数 K 呈线性关系.我们通过污点传播分析,定位潜在问题,消除了不可行的污点传播路径,避免了冗余的验证操作.K 减小了,动态验证对程序性能的影响也就降低了.用户可以自定义优化验证函数的实现,以求更高的精度和效率.

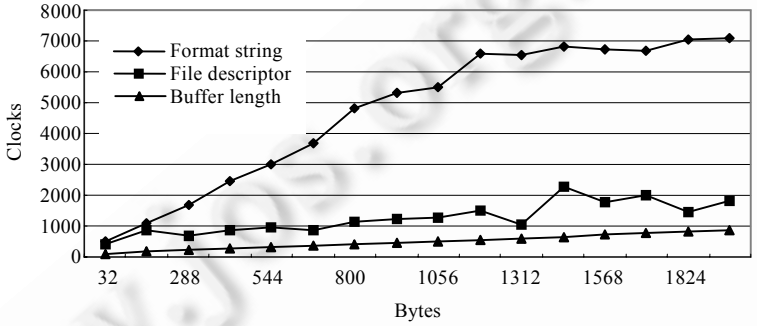


Fig.4 Comparison of runtime costs among instrumented verification routines

图 4 插装验证函数的运行时开销对比

3.2 分析示例

我们测试了文献[22]提到的 wu-ftpd 2.6.0,以找到的一个漏洞来演示污染数据的识别和验证过程.这是一段可能导致格式化串漏洞的代码,涉及两个文件中的 3 个函数.因代码较多,为简洁起见只列出分析相关的部分,如图 5(a)所示.编译器将源代码转换为图 5(b)的 SSA 中间表示之后,变量经过重新命名,控制流结构也发生了一些变化.

首先进行信息流分析,分别得到 *site_exec*,*lreply* 和 *vreply* 函数的内部信息流,如图 6(a)、图 6(b)所示,直接流已绑定了其产生位置.图 6(a)中粗体字表示的两条,前者是根据函数摘要生成的直接信息流,即 *fgets* 函数通过实参传入外部数据,修改了 *buf*_1 指向的缓冲区,此时,*fgets*|1 与 *buf*_1 之间存在双向的信息流动;而后者则是通过变量 *buf*_1 的传递作用得到的间接信息流.后面斜体字的还有由 if 语句中的判断条件产生的隐式信息流.在所有生成的信息流中,常量不会产生或传递污染,没有程序行为实体参与的信息流没有后续分析的必要,故约减时排除了⟨200,*lreply*|1⟩,⟨*D*.0_3,200⟩,⟨*D*.0_3,*buf*_1⟩和⟨4,*vreply*|1⟩.

污点传播分析在上层函数 *site_exec* 约减后的信息流中搜索 *TM* 类型实体,针对外部输入函数及其第 1 个参数的组合“*fgets*|1”展开检测过程.当前没有发现污染数据的使用,所以由⟨*fgets*|1,*lreply*|2⟩启动过程间的判定例程,将 *fgets*|1 映射到 *lreply* 的第 2 个形参 *fmt*_5,如图 6(c)所示.由于 *fmt*_5 的传递作用,将继续映射到 *vreply* 的相应形参 *fmt*_8,并最终在 *vreply* 内得到⟨*fgets*|1,*vsnprintf*|3⟩的间接信息流.其传递路径上的操作改变 *buf*_1 的安全级为“*tainted*”,通过变量 *fmt*_5 和 *fmt*_8 进行传播,并为 *TV* 类型的实体“*vsnprintf*|3”所使用.这满足了第 1.2 节的定义 5,可以判断该信息流传播了污染,应该给出格式化串漏洞的警告.分析程序将间接流传递经过的相邻实体还原为直接流,根据其绑定的产生语句位置映射回图 5(a)中的源代码,输出 *ftpcmd.y*:1930,1935,*ftpd.c*:5343,5353,

5275 和 5290 的代码行号,作为污点传播的路径.最后,还在 *vsnprintf* 之前自动插入了对其第 3 参数——*fmt*_8 的验证函数,检测格式化导向符,保证运行时的安全.

```
ftpcmd.y
1865  void site_exec(char *cmd)  {
        ...
1930     while (fgets(buf, sizeof buf, cmdf)) {
          ...
1935       lreply(200, buf);
          ...
1941     }
        ...
1949  }

ftpd.c
5275  void vreply(int n, char *fmt, va_list ap))  {
        ...
5290     vsnprintf(buf+(n ? 4 : 0), n ?
           sizeof(buf)−4 : sizeof(buf), fmt, ap);
        ...
5306  }

5343  void lreply(int n, char *fmt, ...) {
        ...
5352     /* send the reply */
5353     vreply(USE_REPLY_LONG, n, fmt, ap);
        ...
5356  }
```

(a)

```
void site_exec(char *cmd_0)  {
  ...
  L0: D.0_3 = fgets(buf_1, sizeof(buf_1), f_2);
  if (D.0_3!=0B)
  {
    lreply(200, buf_1);
    ...
    goto L0;
  }
  ...
}

void lreply(int n_4, char *fmt_5, ...) {
    ...
    vreply(4, n_4, fmt_5, ap_6);
    ...
}

void vreply(int n_7, char *fmt_8, va_list ap_9) {
    ...
    vsnprintf(buf_10, D.0_11, fmt_8, ap_9);
    ...
}
```

(b)

Fig.5 Vulnerable code in wu-ftpd 2.6.0 and its SSA intermediate representation

图 5 wu-ftpd 2.6.0 中的脆弱性代码及其 SSA 中间表示

```
void site_exec() {
   ⟨f_2,fgets|3⟩ & 1930,
   ⟨szeof(buf_1),fgets|2⟩ & 1930,
   ⟨buf_1,fgets|1⟩ & 1930,
   ⟨fgets|1,buf_1⟩ & 1930,
   ⟨buf_1,lreply|2⟩ & 1935,
   ⟨200,lreply|1⟩ & 1935,
   ⟨fgets|1,lreply|2⟩,
   ⟨D.0_3,buf_1⟩ & 1930,
   ⟨D.0_3,200⟩ & 1930,
   ⟨D.0_3,lreply|2⟩ & 1930,
   ⟨D.0_3,lreply|1⟩ & 1930,
   ...
}
```

(a)

```
void lreply() {
   ⟨ap_6,vreply|4⟩ & 5353,
   ⟨fmt_5,vreply|3⟩ & 5353,
   ⟨n_4,vreply|2⟩ & 5353,
   ⟨4,vreply|1⟩ & 5353,
   ...
}

void vreply() {
   ⟨ap_9,vsnprintf|4⟩ & 5290,
   ⟨fmt_8,vsnprintf|3⟩ & 5290,
   ⟨D.0_11,vsnprintf|2⟩ & 5290,
   ⟨buf_10,vsnprintf|1⟩ & 5290,
   ...
}
```

(b)

```
⟨fgets|1,lreply|2⟩
»
 void lreply(int n_4,char*fmt_5,...)
»
 ⟨fmt_5,vreply|3⟩ & 5353
»
 void vreply(int n_7,char*fmt_8,va_list ap_9)
»
 ⟨fmt_8,vsnprintf|3⟩ & 5290
»
  ⟨fgets|1,vsnprintf|3⟩
»
  fgets|1↔buf_1→lreply|2→fmt_5
    →vreply|3→fmt_8→vsnprintf|3 ■
```

(c)

Fig.6 Details of information-flow and taint propagation analysis

图 6 信息流及污点传播分析具体细节

4 相关工作及讨论

污点传播分析在脆弱性检测方面已经有了一些应用.静态的,Livshits 等人[23]针对常见 Web 漏洞,配合 PQL (program query language)[24]描述的污点检查策略,在 Java 下实现了污染数据的静态处理.Shankar 等人[22]基于类型系统的相关理论,提出了一种 C 语言程序格式化串漏洞的检测方法.该方法需要添加额外的类型注释(annotation),流不敏感的分析也产生了大量的误报.ARCHER[6]和 IPSSA[25]则采用了路径敏感的分析方法,前者检测程序中的内存访问错误,发现时只报告漏洞位置,并无错误路径的具体描述;后者扩展了 SSA 概念,加入定

值-使用(definition-use)关系检测由输入导致的缓冲区溢出漏洞.其简单地将污染数据的所有使用作为判别依据,而非污染数据和函数以及特定参数的组合,降低了模型的精确程度.相比之下,本文的方法适用于多种程序设计语言,直接分析源代码,不增加用户负担.信息流与污点传播的结合,在信息的传递路径上反映了程序的数据流和控制流特征,便于漏洞的定位和追踪.而且流敏感的分析与完备的污点传播模型可以检测由外部输入数据导致的注入型脆弱性,具有更广的检测范围.PQL 作为一种描述程序代码模式的语言,本文污点检查策略的描述方式与之相比,功能相同却更为简洁明了,易于掌握和后续扩展.

动态方法中,Perl 脚本语言实现了一种称为“污点模式(taint mode)”的安全机制[26],由解释器在执行时进行检测.FlexiTaint[27]通过定制的处理器逻辑和指令操作,将应用程序内存映射到 $L1$ 缓存中的一组污点记录位,执行时同步更新污点映射信息.而 Newsome 等人[28]则使用了影子内存(shadow memory)技术,将每字节内存映射到一个 4 字节的影子内存,由后者指向的数据结构标记出其指示内存的污染状态,记录程序执行时的函数调用及堆栈快照(snapshot)等信息.我们的方法位于编译器层级,不需要额外的处理器结构和指令译码逻辑,信息流分析找出程序实体间的信息传递路径,污点传播只需要为每条信息流建立和维护一个状态变量,具有较小的内存消耗.而且静态分析排除了不会导致脆弱性的程序路径,减少了验证函数的执行次数,为动态分析的开销过大问题提供了一个解决思路.

5 结 论

传统静态分析精确性不高的原因[13]:一是对待分析程序的语义近似不够准确,包括时序属性建模、控制流路径模拟等多个方面;其次是由于动态数据的不确定性,静态过程缺乏对其的实时验证能力,往往只能做最保守的近似估计.信息流分析作为脆弱性检测的有效手段,不局限于特定的环境和数据类型,能够较好地反映程序数据流和控制流特征.而且,信息流可以追溯数据的产生及传递路径,有利于脆弱性的定位和修正.我们将程序转换到保留信息较为完全的 SSA 中间表示,抽取显式和隐式信息流并计算可达性关系,改进了分析模型的精细程度;在污点传播过程中,制定简便而细粒度的污染数据识别规则,随后插入的运行时函数也为数据的动态验证提供了可能.该过程中使用的检查策略和验证函数都是可定制的,具有良好的可扩展性.

信息流和污点传播分析在脆弱性检测方面有着广泛的应用前景.脆弱性检测可以是一个前向或者后向的过程,将来的工作中,我们将结合两个方向分析各自的特点,有针对性地研究适合于各种脆弱性的检测方法.

References:

[1] CVE terminology page. 2009. http://www.cve.mitre.org/about/terminology.html#vulnerability

[2] Sekar R. An efficient black-box technique for defeating Web application attacks. In: Vigna G, ed. Proc. of the Network and Distributed System Security Symp. (NDSS 2009). San Diego: National Security Agency Press, 2009. 23−39.

[3] CVE and CCE statistics query page. 2009. http://web.nvd.nist.gov/view/vuln/statistics?execution=e1s1

[4] Open Web Application Security Project (OWASP). The ten most critical Web application security vulnerabilities. 2007. http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

[5] Lam MS, Martin MC, Livshits VB, Whaley J. Securing Web applications with static and dynamic information flow tracking. In: Hatcliff J, ed. Proc. of the 2008 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation. New York: ACM Press, 2008. 3−12. [doi: 10.1145/1328408.1328410]

[6] Xie YC, Chou A, Engler D. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In: Paakki J, ed. Proc. of the 9th European Software Engineering Conf. Held Jointly with 11th ACM SIGSOFT Int’l Symp. on Foundations of Software Engineering. New York: ACM Press, 2003. 327−336. [doi: 10.1145/940071.940115]

[7] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Ning P, ed. Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2004. 116−127. [doi: 10.1145/1315245.1315261]

[8] Sabelfeld A, Myers AC. Language-Based information-flow security. IEEE Journal on Selected Areas in Communications, 2003, 21(1):5−19. [doi: 10.1109/JSAC.2002.806121]

[9] Denning DE, Denning PJ. Certification of programs for secure information flow. Communications of the ACM, 1977,20(7): 504−513. [doi: 10.1145/359636.359712]

[10] Goguen JA, Meseguer J. Security policies and security models. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society Press, 1982. 11−20. [doi: 10.1109/SP.1982.10014]

[11] Liu Y, Milanova A. Static analysis for inference of explicit information flow. In: Krishnamurthi S, ed. Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE). New York: ACM Press, 2008. 50−56. [doi: 10.1145/1512475.1512486]

[12] Hsieh CS. A fine-grained data-flow analysis framework. Acta Informatica, 1997,34(9):653−665. [doi: 10.1007/s002360050101]

[13] Huang YW, Yu F, Hang C, Tsai CH, Lee DT, Kuo SY. Securing Web application code by static analysis and runtime protection. In: Feldman S, ed. Proc. of the 13th Conf. on the World Wide Web. New York: ACM Press, 2004. 40−52. [doi: 10.1145/988672.988679]

[14] Chess B, West J. Secure Programming with Static Analysis. Boston: Addison-Wesley, 2007. 130−132.

[15] Howard M, LeBlanc D. Writing Secure Code. 2nd ed., Redmond: Microsoft Press, 2002. 53−58.

[16] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. on Programming Languages and Systems (TOPLAS), 1991,13(4):451−490. [doi: 10.1145/115372.115320]

[17] Scholz B, Zhang CY, Cifuentes C. User-Input dependence analysis via graph reachability. In: Antoniol G, ed. Proc. of the 8th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. New York: ACM Press, 2008. 25−34. [doi: 10.1109/SCAM.2008.22]

[18] Pearce DJ, Kelly PHJ, Hankin C. Efficient field-sensitive pointer analysis for C. ACM Trans. on Programming Languages and Systems (TOPLAS), 2007,30(1):105−146. [doi: 10.1145/1290520.1290524]

[19] Heintze N, Tardieu O. Ultra-Fast aliasing analysis using CLA: A million lines of C code in a second. In: Burke M, ed. Proc. of the Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2001. 254−264. [doi: 10.1145/378795.378855]

[20] Stallman RM, the GCC Developer Community. Using the GNU Compiler Collection. Boston: GNU Press, 2008. 90−107.

[21] Stallman RM, the GCC Developer Community. GNU compiler collection internals. 2009. http://gcc.gnu.org/onlinedocs/gccint/Type-Information.html#Type-Information

[22] Shankar U, Talwar K, Foster JS, Wagner D. Detecting format string vulnerabilities with type qualifiers. In: Park Y, ed. Proc. of the 10th USENIX Security Symp. Berkeley: USENIX Press, 2001. 201−220.

[23] Livshits VB, Lam MS. Finding security vulnerabilities in Java applications with static analysis. In: Pai V, ed. Proc. of the 14th USENIX Security Symp. Berkeley: USENIX Press, 2005. 271−286.

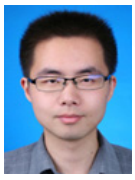
[24] Martin MC, Livshits VB, Lam MS. Finding application errors and security flaws using PQL: A program query language. In: Johnson R, ed. Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). New York: ACM Press, 2005. 365−383. [doi: 10.1145/1094811.1094840]

[25] Livshits VB, Lam MS. Tracking pointers with path and context sensitivity for bug detection in c programs. In: Paakki J, ed. Proc. of the 11th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2003. 317−326. [doi: 10.1145/940071.940114]

[26] Christiansen T. Perl security. 1997. http://www.perl.com/doc/manual/html/pod/perlsec.html

[27] Venkataramani G, Doudalis I, Solihin Y, Prvulovic M. FlexiTaint: A programmable accelerator for dynamic taint propagation. In: Carter J, ed. Proc. of the 14th Int'l Symp. on High Performance Computer Architecture (HPCA). New York: ACM Press, 2008. 173−184. [doi: 10.1109/HPCA.2008.4658637]

[28] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Harder E, ed. Proc. of the Network and Distributed System Security Symp. (NDSS 2005). San Diego: National Security Agency Press, 2005. 187−204.



黄强(1987－),男,云南昆明人,硕士生,主要研究领域为程序分析,信息安全.

曾庆凯(1963－),男,博士,教授,博士生导师,主要研究领域为信息安全,分布计算.